GOBLIN

Manager:	Kevin Xiao, kkx2101
Language Guru:	Bayard Neville, ban2117
System Architect:	Gabriel Uribe, gu2124
Tester:	Christina Floristean, cf2469

INTRODUCTION

Goblin is a language designed to allow anyone to make their own turn-based adventure game without extensive knowledge of software development. It will follow a simplified object oriented model that hews as closely as possible to the familiar way that things act in real life.

All programs result in a turn based game on a rectangular ASCII map. There may be multiple rectangular maps accessed from the initial one via doors, portals, or hatches. The games all follow the same basic loop structure:

Player moves

Print the map

Non-player characters move

Print the map

DESCRIPTION

Goblin is designed with the purpose of creating turn based video games - a video game where the player takes turns versus the computer. The compiled game is designed to run on a command line interface. The program will initiate a redraw of the map after each turn, in this way we can simulate movement. Players will be able to interact with the game using programmer defined commands. Programmers are able to define their game in terms of objects, their behaviors, user commands, and a set of world maps. Maps are square grids of ASCII characters that objects navigate, which means it is possible to supply your own ASCII maps although algorithmic map generation is suggested for more complex maps. Each object shown on a user's screen (the map) is an ASCII character with a specific value, for example a goblin could be represented by 'G'. There are no named instances of entities. Entities of the same type are only distinguished by their xy coords and field values. Every turn the game automatically runs the behavior of each entity that has one, so there is no need to directly reference them, except by relative position.

OPERATORS

Symbol	Action
+, -, *, /	Math operators (Add, Subtract, Multiply, Divide)
and, or, not	Logical operators
!=, ==, <, >, <=, >=	Boolean comparisons (Not equal to, Equal to, Less than, Greater than, Less than or equal to, Greater than or equal to)
=, +=, -=	Assignment (Assign, Assign and increment, Assign and decrement)

COMMENTS

Symbol	Action
//	Single-Line Comment
/* comment */	Multi-Line Comment

BUILT-IN TYPES

Type	Description	Initialization
Integer	Numbers variables.	intName = 5;
String	Text variables.	stringName = “Name”;
Boolean	True / False.	boolName = True;
Character	Character variables.	charName = ‘c’;
Float	Decimal number variables.	floatName = 5.5;
Array	Array variables.	arrayName = [5];

Entity	Any object or character in a game. Automatically has a unique single ASCII character representation, xy coords, and solid = true. May also have other fields and a sequence of behaviors. May inherit fields from another entity using “is”. Fields must have a default value that determines type.	entities {
		    entityName(charSymbol) has {
		        // Fields
		        health = 5;
		    } does {
		        behaviorName;
		    }
		}

Behavior	A set of actions that can be assigned to an Entity which performs them every round of the game.	behaviors {
		    behaviorName {
		        /* Set of conditions and
		           movements specific to
		           each behavior */
		    }
		
		}

Functions	Dynamic operations to allow for code reuse inside of behaviors.	functions {
		    functionName (parameters) {
		        // Action to perform
		    }
		}

World	2D matrix of entities with associated semi-global variables to save world specific state	worlds {
		    worldName {
		        /* Placement functions and
		           world state variables */
		    }
		}

CONTROL FLOW

Type	**Description**	**Structure**

for loop	Iterate consecutively for set amount of turns.	for () {
		}

while loop	Iterate while a boolean variable is true.	while () {
		}

if / elseif / else	Conditional statements.	if () {
		}
		elseif () {
		}
		else {
		}

KEYWORDS

Word	Description
player	Special singleton entity that persists between worlds
is	entity2 is entity1 in entity definition means entity2 inherits from entity1
has	entity1 has { f = 1; } means entity1 has field f with default value 1
does	entity1 has {} does { foo; } means entity1 does foo every turn
when	Used for choosing player behavior based on user input

EXAMPLE PROGRAM

```
player(@) has {
   health = 10.0;
   attack = 1.5;
   money = 0;
   swordtype = 7;
} does {
   player_move_left when “l”;
   player_move_right when “r”;
   player_move_up when “u”;
   player_move_down when “d”;
   player_attack when “a”;
}
entities {
   monster(m) has {
      health = 1.0;
      attack = 1.0;
   }

   goblin(g) is monster has {
      health = 3.0; // overrides health = 1.0 from monster
      // inherits attack = 1.0 from monster
      money = 1; // adds new money field
   } does {
      handle_death;
      chase_player;
   }
```

```
    ghost(~) has {
        solid = false; // solid = true is default unless specified to be false
    } does {
        wander;
    }

    rock(O) has {
        durability = 10.0;
        stone = 3; // Player could receive this when breaking stone
    }

    tree(T) {
    }

    grass(.) {
        solid = false;
    }

    spike(^) has {
        symbol = ‘T’
        solid = false;
        attack = 1.0;
    } does {
        dangerous_floor;
    }

    lava(#) has {
        solid = false;
        attack = 13.5;
    } does {
        dangerous_floor;
    }
}
behaviors {
    chase_player {
        if distance(me, player) < 10 {
            move_towards_player(1);
        }
        if adjacent(me, player) {
            attack(me, player);
        }
    }
    handle_death {
        if me.health <= 0 {
            player.money += me.money;
        }
    }
}
functions { // for general purpose function reuse
    attack(monster attacker, player defender) {
        defender.health -= attacker.attack;
    }
    distance(entity a, entity b) { // this would probably be in the std library
        distance = 0;
        if(a.x > b.x) {
            distance += a.x - b.x;
        } else {
            distance += b.x - a.x;
        if(a.y > b.y) {
            distance += a.y - b.y;
        } else {
            distance += b.y - a.y;
        }
        return distance;
    }
}
worlds {
    house {
        goblins_defeated = 0 // Variable to keep world-specific state
        new_world(5, 10); // built-in function to make blank world
        place(goblin, 1, 2); // built-in function to place entities
        place(player, 4, 9);
    }
}
```